

## PREDICCIÓN DE SALTO (Simple)

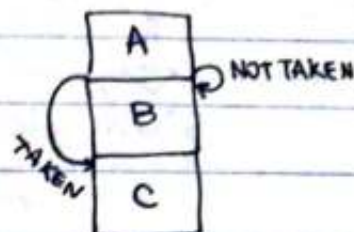
### ⊗ Predicted NOT TAKEN

Este predictor asume que el salto nunca se toma, es decir que la siguiente ~~instrucción~~ instrucción se encuentra en  $PC = PC + 4$ .

Funciona adecuadamente en los casos como:

- #1 instrucción branch
- #2 instrucción sucesora secuencial
- #3 instrucción destino para branch taken

es decir para saltos hacia adelante.

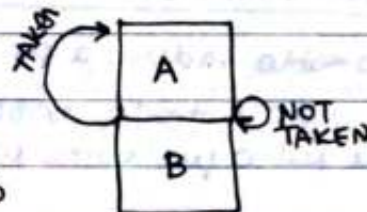


### ⊗ Predicted TAKEN

El procesador asume que el salto siempre se toma, continúa el fetch ~~x~~ el target del salto condicional.

Es útil en el caso de los loops, ya que en general estos se van a tomar muchos veces menos la última. Son siempre saltos hacia atrás.

- inst. destino para branch taken
- instrucciones a ejecutar iterativamente
- instrucción branch



Entonces si se realiza profiling para los saltos hacia adelante, sería útil predecir { NOT TAKEN en los saltos hacia adelante  
TAKEN en los saltos hacia atrás

### • Otra potencial solución es DELAYED BRANCH

La idea es llenar el tiempo desde el fetch del branch hasta su resolución con otras instrucciones independientes que no se ven afectados por el branch.

Esto se puede hacer en pipelines muy cortos o en los casos en los que la resolución del branch es rápida, si no es difícil encontrar instrucciones que sirvan para llenar los Delay Slots.

Se consigue reordenando el código adecuadamente en tiempo de compilación.

- LOOP UNROLLING Cuando se tiene un loop (por ej. un for-loop) donde sus instrucciones son paralelizables (independientes) se puede ir desenrollando los ciclos de forma que finalmente haya menos saltos y condiciones que chequear. De esta forma desaparecen potenciales riesgos de penalización.

Se puede usar code profiling para que los saltos sean los más probables.



Por ejemplo (en C) si se quiere aplicar la función  $f$  a los elementos del vector  $V$  y las aplicaciones son independientes:

```
for (int i=0; i<256; i++) {
    f(V[i]);
}
```

↓  
256 iteraciones

↻  
loop  
unrolling

```
for (int i=0; i<256; i+=4) {
    f(V[i]);
    f(V[i+1]);
    f(V[i+2]);
    f(V[i+3]);
}
```

↓  
 $\frac{256}{4} = 64$  iteraciones.

### PREDICCIÓN DINÁMICA DE SALTOS

Todos los métodos anteriores son estáticos y tienen que ver con la forma de ordenar el código del compilador.

#### BPB : Branch Prediction Buffer (Last Time predictor)

Es una tabla indexada por la dirección de memoria de la instrucción del branch, y para cada entrada contiene un bit que indica si la última vez que se lo vio este salto fue tomado o no.

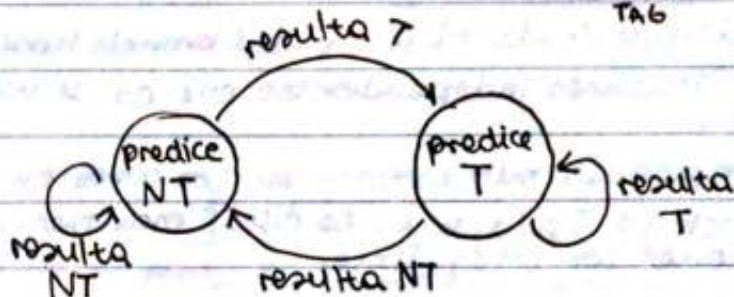
PC [index | TAG]



TAG

último resultado  
0 → NOT TAKEN  
1 → TAKEN

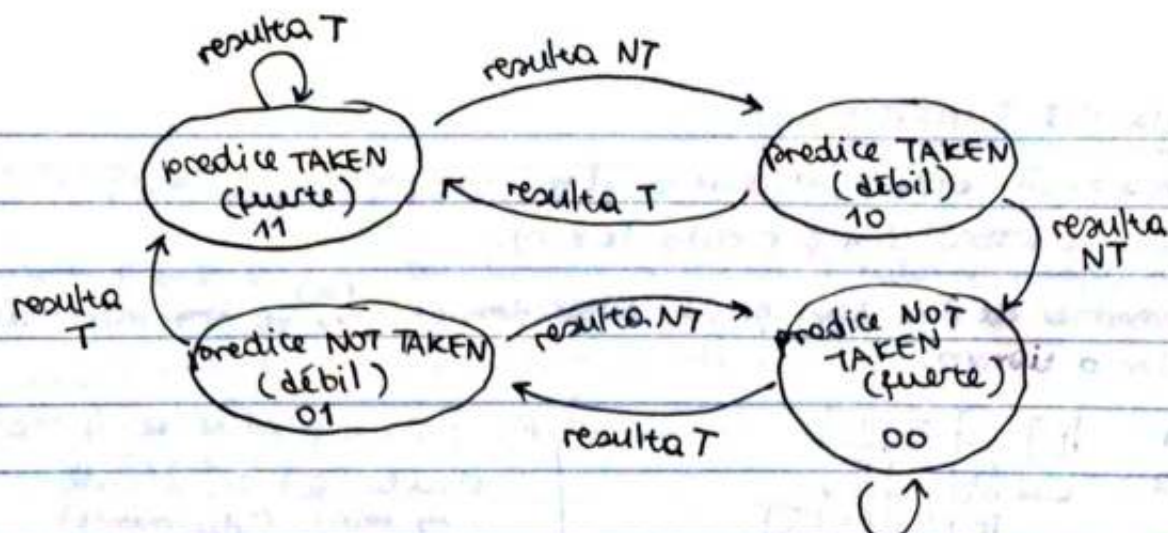
hay varios formas de implementarlo podría no usarse un TAG y que la info no sea segura que provenga de la misma branch x ejemplo (pero ocupa menos espacio).



Este predictor cambia muy rápido de opinión. Además falla 2 veces en los ciclos y es malo si el resultado es alternante.

Se puede mejorar con una predicción de dos bits:





De esta forma dos resultados iguales consecutivos son necesarios para cambiar la predicción.

### IMPLEMENTACIÓN

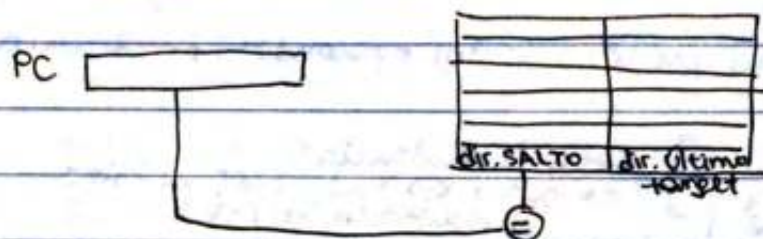
Se implementa en la etapa de fetch **[E]** del pipeline, como un pequeño caché de direcciones. Sirve para saber que una instr. es un branch antes de decodificarla **[D]**.

también se puede implementar directamente en el caché de instrucciones aprovechando dos bits.

Este método tiene una eficiencia superior al 82%, mayor x mucho a la de 1 bit. No hay mejoras significativas por aprovechar más bits de predicción.

### BTB: Branch Target Buffer

Es un caché de instrucciones de salto que para cada entrada guarda el par: (dirección de salto / dirección de target resuelta), no los bits de T/NT.



~~Se puede implementar directamente en el caché de instrucciones:~~

Se puede usar con/sin un predictor para tener predichados los direcciones posibles luego del salto.



## PROCESADORES SUPERESCALARES

Aprovechan más paralelismo dentro del mismo procesador para ejecutar más de una instrucción por ciclo de reloj.

Superscalar de dos vías: puede realizar dos etapas<sup>(=)</sup> de dos instrucciones al mismo tiempo:



ejemplo con pipeline de 4 etapas,  
ejecuta 2 intr. x ciclo de reloj (idealmente).

Pueden ser sólo paralelizables algunos etapas (x ejemplo usar más de una ALU).

→ tiempo.

El aumento en paralelismo aumenta el riesgo de encontrar obstáculos, a la vez que la penalización de un pipeline flush o un stall es más grande.

## EJECUCIÓN FUERA DE ORDEN

En el caso de poder ejecutar varias instrucciones a la vez, un PIPELINE STALL generado x ejemplo por esperar un dato de memoria, detiene todo el pipeline.

Nos gustaría tomar la filosofía de DATA FLOW y ejecutar las instrucciones a medida que sus operandos están listos, y no estar esperando a causa de dependencias / stalls que no son ciertos sino forzados por la ejecución en orden.

Por ejemplo, dos operaciones con los mismos operandos no deberían tener que esperarse entre sí:

DIV  $R_1 \leftarrow R_2, R_3$   
ADD  $R_4 \leftarrow R_1, R_3$   
SUB  $R_6 \leftarrow R_5, R_7$

esta dependencia de datos es real (ADD necesita el resultado de DIV)

esta dependencia de datos no es real, el SUB podría ejecutarse antes y mantener la misma semántica del programa.

(Acá, DIV es una op. larga de varios ciclos)

la idea es despachar las instrucciones cuando sus operandos están listos.

Las instrucciones son analizadas luego de su decodificación [D] que debe seguir siendo en orden para preservar la semántica.



Si por ejemplo podemos paralelizar la ejecución de las siguientes instrucciones:

MUL  $R_3 \leftarrow R_1, R_2$   
ADD  $R_3 \leftarrow R_3, R_1$   
ADD  $R_4 \leftarrow R_6, R_7$   
MUL  $R_5 \leftarrow R_6, R_8$   
ADD  $R_7 \leftarrow R_3, R_5$

considerando un MUL de 4 ciclos y un ADD de un ciclo, se compone ejecución en orden con fuera de orden:

En orden (con forwarding)

#1 F D E E E E W  
#2 F D STALL E W  
#3 F STALL D E W  
#4 F D E E E E W  
#5 F D STALL E W

Fuera de orden:

#1 F D E E E E W  
#2 F D WAIT E W (tiene que esperar la disponibilidad de  $R_3$ )  
#3 F D E WAIT W (tiene que esperar a que se lea  $R_3$ )  
#4 F D E E E E W  
#5 F D WAIT E W (tiene que esperar a  $R_5$ ).

esto es en  
el caso  
de fuera  
de orden

~~\* Se agregan los tipos de dependencias de datos, que pueden ser:~~

RAW (Read After Write) → esto pasa siempre (true dependency) y causa pipeline stalls cuando se ejecutan en orden.

DIV  $(R_1) \leftarrow R_2, R_3$   
ADD  $R_4 \leftarrow (R_1), R_2$

Se refiere a cuando una instrucción posterior lee un operando escrito por una operación previa. El riesgo es si la lee antes de que el otro la escriba, obteniendo así un dato incorrecto. Es una dependencia real ya que la 2da operación necesita el dato calculado por la primera.

WAR (Write After Read)

DIV  $R_1 \leftarrow (R_2), R_3$   
ADD  $(R_2) \leftarrow R_4, R_5$

Como podemos despachar instrucciones fuera de orden, en casos como estos hay que ver que  $R_2$  sea leído antes de ser modificado por la instrucción posterior.



## WAW (Write After Write)

DIV  $(R_1) \leftarrow R_2, R_3$

ADD  $(R_1) \leftarrow R_4, R_5$

En estos casos se debe asegurar que la escritura se haga en orden, o que al menos sólo ocurra la 2da (Ambos en el caso de excepciones precisas).

Todo esto se soluciona si ~~se ejecutan~~ escribimos en orden, aunque ejecutemos en paralelo (~~se ejecutan en paralelo~~). (paralelo).  
 Tomasulo + ROB

## EXCEPCIONES IMPRECISAS

Un problema posible al ejecutar fuera de orden sería la imposibilidad de saber qué instrucciones terminaron y cuáles no, por la posible diferencia entre el orden de la ejecución y el código que fue pensado para ejecución secuencial. Esto es un problema para el programador.

Hay dos posibles motivos a la excepción imprecisa:

- El pipeline completó instrucciones posteriores a la que produjo la excepción.
- El pipeline no completó alguna instrucción anterior a la que produjo la excepción.

En estos dos casos el estado del sistema no es consistente con ~~lo~~ que esperaríamos.

disclaimer:

SCOREBOARDING → no entendí cómo era ni por qué es  $\neq$  a Tomasulo.

Es el método más sencillo para implementar ejecución fuera de orden evitando los riesgos asociados: WAR y WAW. en orden sec.

Las instrucciones pasan por los siguientes etapas; luego de ser decodificados:

~~① Unidad de envío: Si hay una unidad de ejecución libre y no hay ninguna otra instrucción que necesite el mismo operando de destino, se la envía a la unidad de ejecución.~~

① Chequeo de riesgos: ~~El~~ El sistema guarda la información de qué registros van a ser leídos/escritos por la instrucción.

Cuando hay instrucciones activas que van a escribir al mismo registro, o no hay unidades funcionales disponibles, se produce un stall (para esa instrucción).

→ para evitar WAW.







Los otros tres principios de Tomasulo (mantener a las instrucciones esperando y despacharlas cuando sus operandos están listos) son tarea de los Reservation Stations.

Es hardware con registros internos que guardan las instrucciones en espera. Cada operando cuyo valor no está disponible posee un TAG que corresponde con el renombre en la Register Alias Table.

Cada vez que una unidad de ejecución hace disponible un operando, su tag y su valor se broadcastean a todas las reservation stations. De esta manera se hace efectivo el register renaming, ya que si estabas esperando un valor previo de un registro, éste tiene otro TAG.

Si un operando destino tiene múltiples escritores, se aplica la última.

Cuando en la reservation station se detecta que una instrucción tiene todos sus operandos listos, se la despacha a la unidad funcional correspondiente. Si no hay ninguna, va en cola en espera.

COMMON DATA BUS: el medio por el cual se realizan los broadcasts de resultados entre los reservation stations y la register alias Table.

#### ⊗ ALGORITMO DE TOMASULO:

IF (RS tiene lugar disponible):

se inserta la instrucción en RS

ELSE:

stall

se tocan sus operandos de la RAT

~~se tocan sus operandos de la RAT~~

se invalida el registro destino y se pone el nuevo tag de la entrada de la RS correspondiente en la RAT que y del registro destino

WHILE (la instrucción está en RS):

la instrucción:

- Mira el tráfico en el CDB en busca de los TAGS que correspondan a sus operandos
- cuando lo detecta lo guarda en su lugar de la RS
- Cuando ambos operandos están ~~capturados~~ capturados sus resultados, la instrucción se marca READY para ser despachada.

IF (Unidad Funcional disponible):

se despacha la instrucción a dicha UF

IF (finalizada la ejecución):

la UF arbitra el CDB

Hace broadcast del valor resultado y su TAG correspondiente.



~~El problema es~~

En la Register Alias table y en las ~~las~~ Reservation Station:

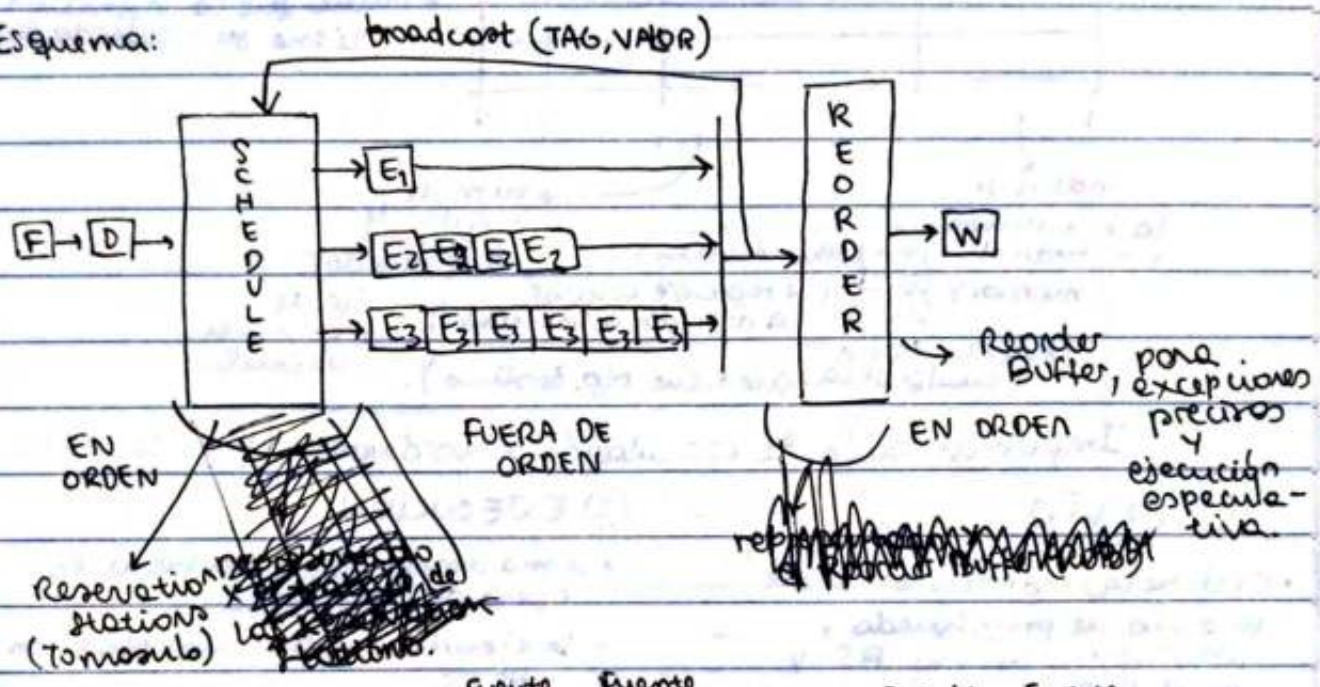
if (tag = tag del broadcast):

valor = valor del broadcast

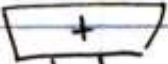
bit de validez = 1

Se invalida ese tag en todo el sistema.

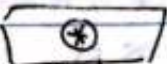
Esquema:



Register Alias Table			Fuente 1		Fuente 2		Fuente 1		Fuente 2			
	V	TAG	VALOR	V	TAG	VALOR	V	TAG	VALOR	V	TAG	VALOR
R0												
R1												
R2												
R3												
R4												
:												



TAG VALOR



tag valor.

### Reorder Buffer (ROB)

Si queremos excepciones precisas usamos la RAT como una fuente fiable. También puede ser el caso de ejecución especulativa, con algún tipo de branch predictor.

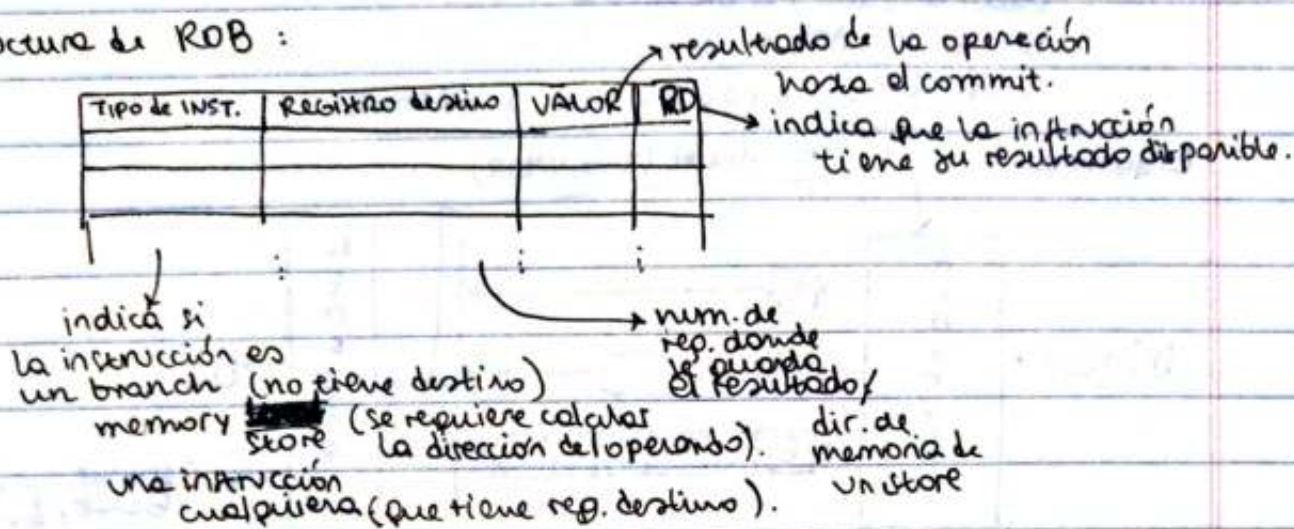
Los registros arquitecturales (los que se miran en una excepción) tienen que ser actualizados x una instrucción cuando esta es la más vieja en el sistema, y no antes.



El ROB aprueba copias de los registros en los cuales va almacenando los resultados de las instrucciones especulativas.

Estos resultados permanecen en el ROB hasta que se tenga la confirmación de que es correcto ejecutarla (COMMIT).

Estructura de ROB :



Implementación de especulación x hardware: (Branches + Obo)

### ① ENVÍO

- Obtiene las instrucciones desde una cola de prebúsqueda, ubicándolos en una RS y en el ROB.
- Si no hay lugar en alguno de produce un STALL.

### ② EJECUCIÓN

- Se monitorea el bus en busca de operandos faltantes.
- Se ejecutan operaciones ready en la RS.

### ③ WRITE RESULT

- Se escribe el resultado al ROB
- Se escribe el resultado a los RS que lo estuvieran esperando.

### ④ COMMIT (de la última int. de ROB).

- Si era un branch incorrecto se flushes el ROB y comienza de nuevo.
- Si era todo correcto se escriben los últimos instrucciones en los registros / memoria. Redes.

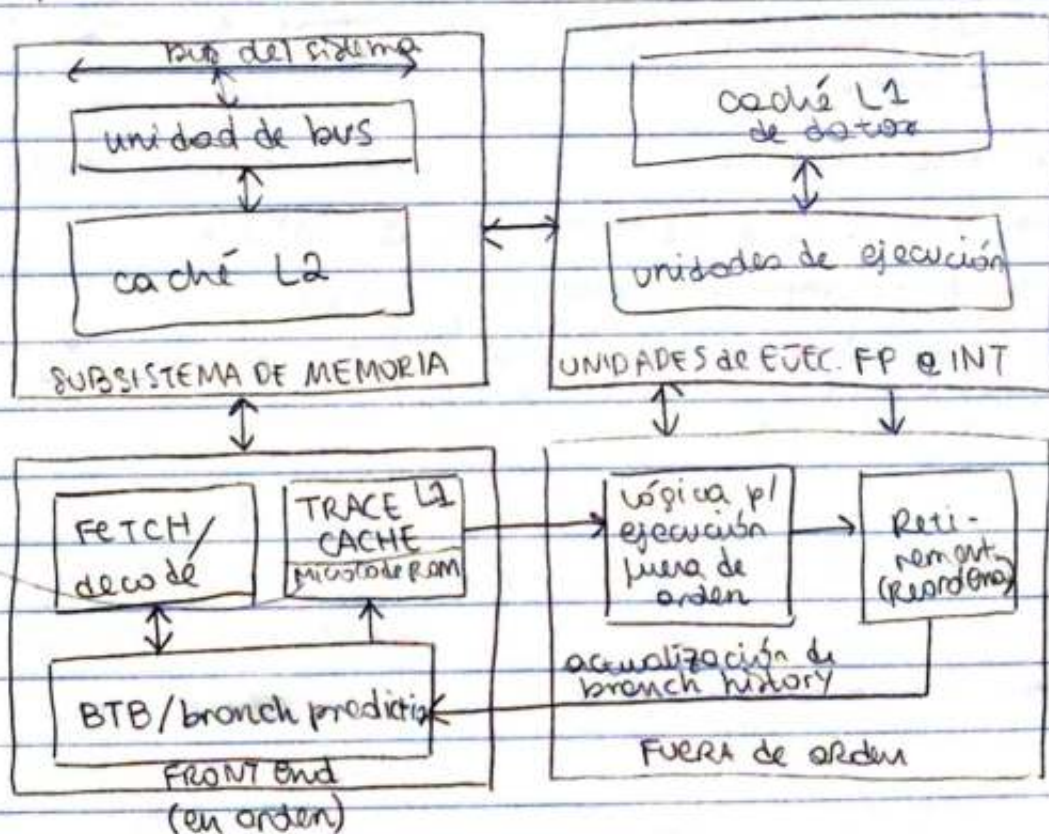


Pentium 4: sucesor de Pentium III.

- va armando los  
trazos con la info  
del branch principal

- Agrega SSE2 (más instrucciones SIMD).

Formas de interacción de interacciones complejas



⑧ FRONT END: La parte que se encarga de hacer FETCH, DECODE de las instrucciones y las prepara para ser usadas más tarde. Le provee instrucciones decodificadas al motor fuera de orden, de acuerdo a su predictor de saltos.

Usando el predictor se ven instrucciones del cache L2.

entre la lógica de decodificado y el motor de ejecución se encuentre el Trace cache, un cache L1 sólo de código que guarda instrucciones ya decodificadas.

Le esta forma se busca que cu se devolva solo una vez, cuando hay un cache miss sobre el TRICE CACHE L1 y hay que buscar en el L2



## LÓGICA DE

⊕ EJECUCIÓN FUERA DE ORDEN: donde las instrucciones son preparadas para su ejecución.

Implementa ejecución fuera de orden p/aumentar la eficiencia.

La lógica de Reordenamiento guarda las operaciones completadas y las va escribiendo en orden secuencial para preservar el estado de la arquitectura, además garantiza la corrección de excepciones precisas.

También reporta información de los branches al predictor.

⊕ UNIDADES de EJECUCIÓN:

Unidades de FP e INT, incluye los registros y está cerca del caché de datos usado en los load/store.

⊕ SUBSISTEMA de MEMORIA:

Incluye el caché L2 y el bus del sistema.

Este caché es compartido para datos/código.

Se comunica con la memoria principal a través del bus externo del sistema (y a los recursos de E/S).



## ② Hyper-Threading

Convierte un procesador físico en dos lógicos.

Estos procesadores lógicos comparten recursos y tienen el estado de la arquitectura duplicado.

Desde el punto de vista de la arquitectura se pueden scheduler procesos o threads a uno o a otro procesador.

Desde el punto de vista de la microarquitectura para duplicar que las instrucciones de ambos se ejecuten usando los mismos recursos.

El objetivo es: una nueva técnica para ganar performance.

- Cada vez hay más programas con threads paralelizables.
- Son comunes los sistemas multiprocesador para lograr más performance.

OTRAS TÉCNICAS SON:

- CMP: Chip Multi Processing: dos procesadores en el mismo chip.
- Time-slice multithreading: Simular multi procesamiento cambiando entre threads de un solo procesador.

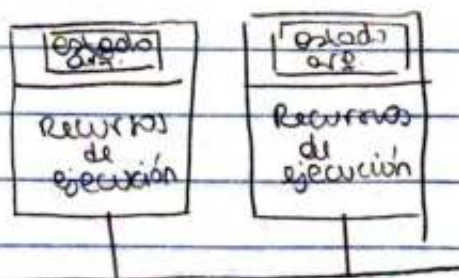
- Multi-threading simultáneo:

Múltiples threads ejecutan en un solo procesador sin switchear, ejecutando al mismo tiempo.

Maximiza performance vs. # de transistores y consumo

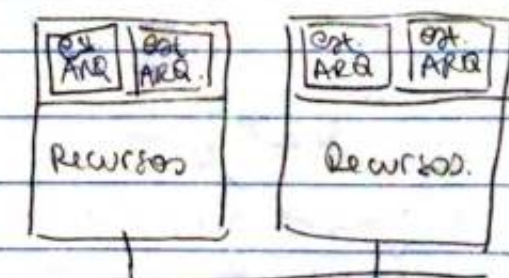
tecnología  
Hyperthreading

Procesadores sin  
HYPER THREADING



2 procesadores físicos  
2 " lógicos

Procesadores con  
HYPER THREADING

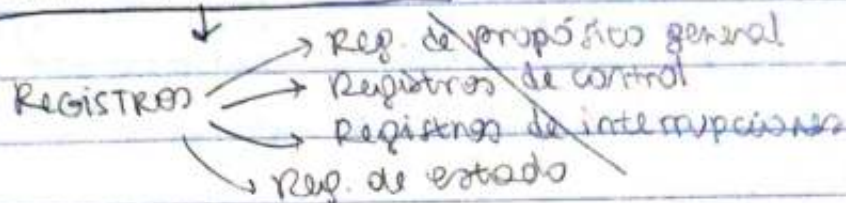


2 procesadores físicos  
4 " lógicos.



- Primera familia q/hyper threading: Intel Xeon.
- Agregando poco hardware / consumo consigue muchos beneficios de performance.

- Dos copias del ESTADO de ARQUITECTURA en c/procesador



- Los procesadores lógicos COMPARTEN todo lo demás.

(cachés, unidades de ejecución, predictores de salto, buses...)

- Cada procesador controla sus propias interrupciones

Metas en la implementación de Intel Xeon

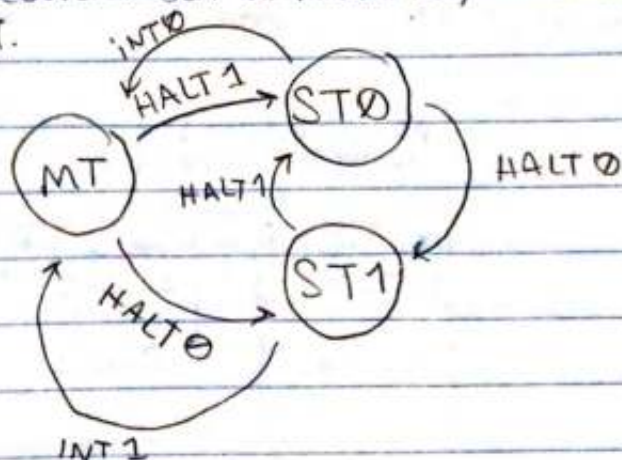
- ⊗ Minimizar el costo en área de chip p/ implementar hyper threading.

- ⊗ Asegurar que cuando un núcleo está STALLeado el otro pueda continuar ejecutando.

- ⊗ Permitir que un <sup>solo</sup> thread pueda correr a la misma velocidad que en un procesador sin hyper threading.

Modos SINGLE-TASK Y MULTI-TASK:

Es posible "apagar" uno de los procesadores lógicos y usar todos los recursos con el restante, mediante la instrucción privilegiada HALT.



- El SO puede apagar un core lógico que no se está usando.

- El SO puede tratar de usar primero  $\neq$  cores físicos antes de pasar a usar los lógicos.



### ③ Pentium M (no basado en Pentium 4 sino en Pentium III).

Es un procesador diseñado p/dispositivos móviles (laptop?) con énfasis en lograr performance ahorrando batería.

- mejor performance vs. más duración de la batería

Pentium M  $\rightarrow$  10% del consumo de energía del sistema.

el resto se porta en otros componentes (pantalla, disco, memoria...)

Des criterios:

Maximizar performance dentro de los límites energéticos

$$E \propto CV^2 F$$

$$F \propto V$$

F: frecuencia

energía consumida  $\propto CV^3$

C: capacidad

V: voltaje

$$\text{performance} \approx \text{IPC} \cdot F$$

↓  
instrucciones por ciclo.

Incrementando el voltaje 1% se incrementa la performance 1% a través de un incremento en frecuencia. Esto resulta en aprox. 3% más energía consumida.

Esto sirve para evaluar si una determinada técnica para aumentar performance es útil o no en términos de energía, o si simplemente es más barato aumentar el voltaje / frecuencia.

Minimizar la energía x tarea

Balace entre ejecución lenta por más tiempo o ejecución intensiva en paralelo, durante menos tiempo.

+ Se mejoró el branch predictor de Pentium 4 y además porte menos energía.

+ loop detector

+ Indirect branch predictor.

}  $\rightarrow$  Global control flow history.

+ Fusión de microoperaciones (usados en ejec. fuera de orden).

en algunos partes de la ejecución pero no en otras

para perder menos energía en la ejecución fuera de orden.



#### ④ Pentium 4 de 90nm

- Cachés más grandes
- Mejores algoritmos (?)
- Hyper-threading
- Mayor frecuencia que el Pentium 4 original

Mejoras en predicción de saltos (estáticos cuando no hay info en BTR)  
+ agregó el indirect branch predictor de Pentium M

...

#### ⑤ Intel Core Duo = Basado en Pentium M

= Procesador móvil.

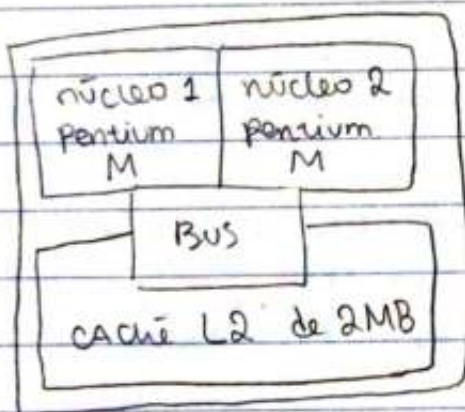
= Usa CMP (Core Multi Processor)

= Más performance con menos consumo.

[no tiene mejoras de performance a nivel de 1 núcleo, sino que todo se basa en mejorar CMP]

La idea es que no es eficiente aumentar la eficiencia de un solo thread, sino que es mejor ver la forma de usar el paralelismo de threads para ganar eficiencia.

Diseño:



Principios:

- Mantener la eficiencia single thread
- Aumentar la eficiencia multi-thread
- Mantener igual el consumo de energía que en Pentium M.
- Que se pueda usar en muchos dispositivos con distintos thermal envelopes (distintas formas de disipar calor).

Cada núcleo tiene una unidad de control térmico.

→ sensores de temperatura para regular niveles de consumo.